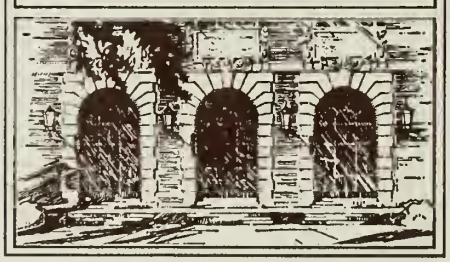


UNIVERSITY OF ILLINOIS

GRADUATE COLLEGE

DIGITAL COMPUTER LABORATORY

REPORT NO. 93

# ORGANIZATION OF A VERY HIGH-SPEED COMPUTER

by

Donald B. Gillies

This work was supported in part by the
Atomic Energy Commission and the Office of Naval Research
under AEC Contract AT(11-1)-415.

# ORGANIZATION OF A VERY HIGH-SPEED COMPUTER

## 1. GENERAL SPECIFICATIONS

The discussion of this type of computer is begun by listing the equipment required:

### 1.1 Input-Output and Auxiliary Storage

one 7-hole photoelectric paper tape reader with a speed of between 1,000 and 2,000 characters per second,

one 7-hole paper tape punch with a speed of 200 to 300 characters per second,

one medium speed printer with at least 60 spaces per line and at least 3 lines per second,

one low-speed printer: 10 or 15 characters per second,

one cathode ray oscilloscope output, with a camera with fast film advance, and a plotting rate of at least 25 k.c.,

at least 4 magnetic tape units with start/stop time of 5 ms or less, approximately one megacycle digit rate, and the ability to read information which has just been recorded,

one magnetic drum with a capacity of 40,000 to 65,000, 52-bit words, a word rate of 6 μs or less (probably by using NRZ recording and parallel storage of words, one track per bit),

provision for connecting other devices to the computer if a need for them is found.

### 1.2 Main Memory

Two 4096 word destructive readout core memories are used in parallel. The cycle time for reading and writing is about 2 μs. The first 0.8 μs of a read

or write operation accounts for the readout, and a further enforced 1.2 μs is required to regenerate the word. Two memory registers called $Z_1$ and $Z_2$ are used for writing and regeneration. When a number to be written has been placed in either $Z_1$ or $Z_2$ the memory will autonomously write the word.

## 1.3 Word Length

The word length is 52 binary digits.

## 1.4 Arithmetic Unit

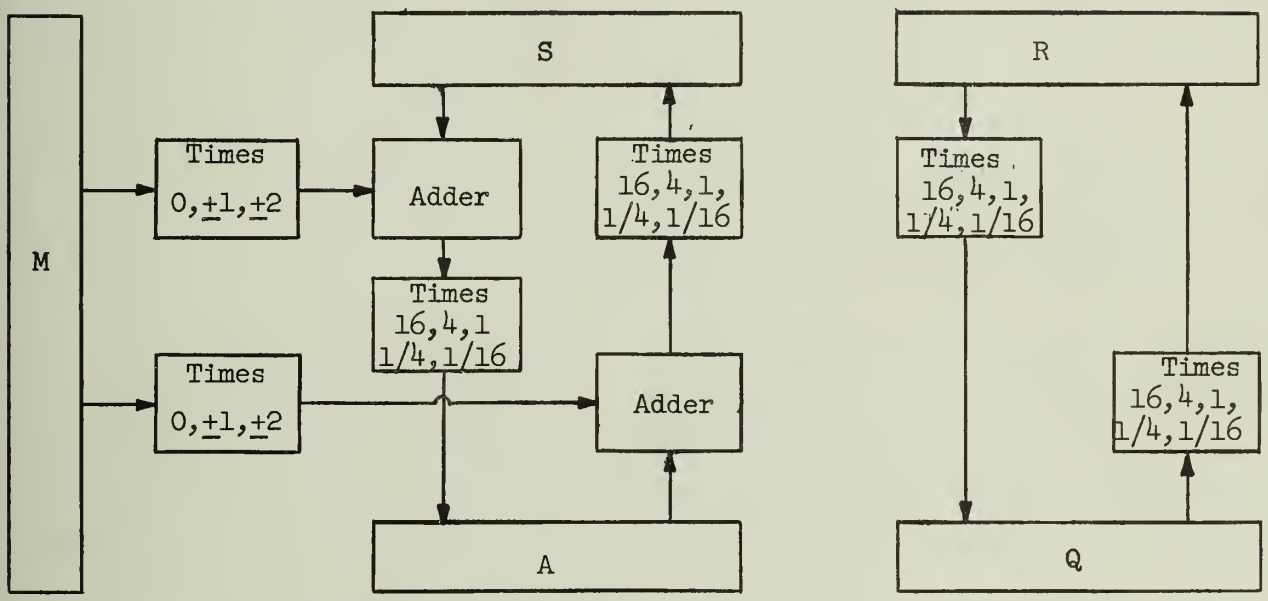
The arithmetic unit may be considered to be in two parts - the main arithmetic unit (MAU) and the exponent arithmetic unit (EAU). The main arithmetic unit deals with the fractional parts of numbers (fixed or floating) while the EAU computes the exponents of arithmetic results, and determines the number of shifts necessary during arithmetic.

A floating point word consists of a 45-bit two's complement fraction f, and a 7-bit integer exponent e, also in two's complement representation. The shift operation in the MAU moves a number in the accumulator left or right by an even number of binary places. Since 2 bits comprise a base 4 digit, shifts are performed in base 4, and e is considered a base 4 exponent. A fraction f is called normalized if it lies in the range 1/4 to 1 in magnitude (excluding -1/4 and +1). A floating point word has the value $f \times 4^e$ where $-64 \leq e \leq 63$. Except for the number zero, f is normalized if the word is held in a storage register.

The main arithmetic unit consists of a double length (92 bits) double rank shifting register, called the accumulator, divided into 2 halves (A,S), (the more significant), and (Q,R), (the less significant), together with a single rank register M which holds an operand. One additional bit per base 4 digit in both A and S holds a stored base 4 carry. Thus the representation of a number in say A and Q (the lower rank of the double length accumulator) consists of a more significant half with one carry bit per base 4 digit and a binary less significant half. A number is called assimilated if it is known that all stored carry digits are zero. When a number is to be stored in the memory from the accumulator,

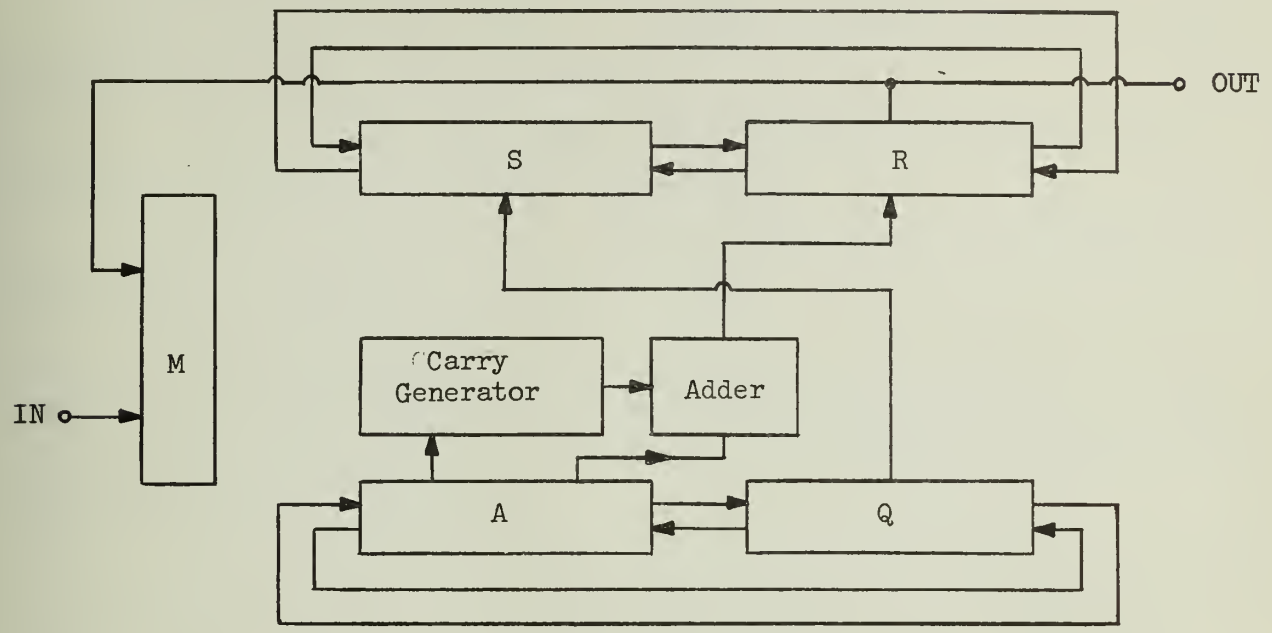
the accumulator is assimilated, normalized, and (usually) rounded. The resulting 45 bits are then combined with the exponent from the EAU and stored as a word in the memory.

There are two adders in the MAU. One adder forms the sum (in stored carry representation) of the number in A plus or minus M or twice M or zero. The result may be gated to S either directly or displaced one or two base 4 digital positions left or right. In a similar manner, another adder forms the sum of the number from S and a multiple of the number from M. The result, possibly shifted, is placed in A. The connections so far described are shown in the following diagram:




End connections between the least significant end of (A,S) and the most significant end of (Q,R) shifted left or right as one double length register. On right shift this requires a partial assimilation of carry digits, since Q and R do not have carry storage. End connections between the most significant end of (A,S) and the least significant end of (Q,R) allow the accumulator to be shifted circularly, left or right. To assimilate the numbers in A, another input to the A-adder (instead of M) is provided by a special "carry generator." The sum digits of the output of the adder will then represent the assimilated value of A. Cross connections allow A (preferably assimilated) to be gated directly to R, and allow Q to be gated to S. Finally R may be gated to M, or

to the fast access memory. These additional connections are represented systematically by the following diagram:




During multiplication, the partial products are generated in (A,S) and shifted right into (Q,R), while the multiplier digits are sensed at the right hand end of (Q,R). The accumulator is normally considered to hold one number (possibly double precision) at the end of each arithmetic instruction. Therefore, the first step in multiplication consists of assimilation, round-off, and transfer of the result from A to R to serve as the multiplier.

During division, the number in M is divided into the double precision number in (A,Q), and quotient digits are inserted at the right end of Q. Finally, the rounded quotient is transferred from Q via S to A, and Q is cleared to zero.

Addition forms the correct (or correctly rounded) sum of the possibly double precision contents of the accumulator, and the single precision number in M. Double precision multiplication and division are comparatively easy to program given this single length to double length add operation. For multiple precision, another add operation is used, which adds provided the double precision result does not have round-off.

## 1.5 Fast-Access Registers

Interposed between the arithmetic unit and controls, on the one hand, and the core memories on the other, are a number of transistorized fast access registers (probably a flow gating memory):

4 data registers $(DR)_1$ $(DR)_2$ $(DR)_3$ $(DR)_4$ used for buffering data between the arithmetic unit and the core memories,

4 control registers $(CR)_1$ $(CR)_2$ $(CR)_3$ $(CR)_4$ used for buffering data between the core memory and controls,

8 temporary storage registers, $R_0$ $R_1 \ldots R_7$, which can be set from the core memories or the arithmetic unit. Each of these registers is subdivided into 3, 17-bit modifier registers which can be used for counting and the formation or modification of addresses.

The execution of a typical arithmetic instruction requires 6 references to this flow gating memory. The speed of this memory is about 0.2 μs per complete cycle; however it is possible to read from one word while writing into another.

## 1.6 Modifier Arithmetic Unit

Address modification and counting is performed concurrently with arithmetic in a separate, 17-bit arithmetic unit which communicates with the modifier in the instruction buffer registers, the shift counter, the control counter, and the core memory address generator.

## 1.7 Controls

For the efficient utilization of an arithmetic unit of such high capability (addition 1μs, multiplication 3μs), it is desirable to use the core memory and arithmetic unit concurrently, anticipating data requirements (both operands and instructions) and, hence, overlapping the memory time of some instructions with the arithmetic time of others. Therefore, each instruction is executed by two controls, called advanced control, and arithmetic control. The function of advanced control is to scan instructions not yet executed by

arithmetic control, perform those operations which do not depend on an arithmetic result, determine what data are required from the memory, and acquire them. For example, during an average multiplication, there is time to read several words from the core memories. Ideally, the memories and the arithmetic unit would both operate continuously, and the over-all speed of the computer operating in this parallel mode would be three times that of a sequential machine with the same number of extra registers. In practice, one control will sometimes have to wait for the other, so the ratio of speeds is not as favorable as 3:1. One control will have to wait for the other if one of the following situations occurs:

a) its operation requires the use of equipment currently being used by the other control;

b) its correct operation depends on a result not yet obtained by the other control;

c) there is no space among the fast access registers to hold more data. Interlocks will be provided to detect these cases.

For problems with large data requirements, it is also desirable to be able to use the drum and one or two tape units for transfers to and from the core memory, without holding up the calculation in progress except at those times when the core memory is required to send a word to an auxiliary storage device. This parallelism is advantageous because of the high expected duty cycle of auxiliary storage devices, and their moderately high data rates. In other words, the rate of transmission of data (6 μs for the drum, and perhaps 50 μs for a tape unit) is not high enough to justify using the computer solely for performing the transfer in question, nor is it low enough that the actual program can be interrupted whenever transfer of one word is required, and an auxiliary program perform the transfer.

Therefore, an additional control of simpler design and with a lower speed requirement than arithmetic control and advanced control is required to execute block transfers of data between the core memory and drum and between the core memory and any two magnetic tape units. Whether this will consist of three controls, or of one control time-shared between the various units, perhaps

using the B-line arithmetic unit, is not yet decided. However, functionally, it will be equivalent to three controls--one for the drum and two which can be switched to any two magnetic tape units.

To reduce the average access time to the drum, each drum order will transfer a number of blocks. The blocks will probably be of 128 words.

The main interconnections are drawn in the diagram on the following page.

## 1.8 Other Facilities

### 1.8.1 Program Interrupt

A register, called the interrupt register, consisting of between 15 and 30 flipflops which tell the states of various parts of the machine will be provided. These are divided into two groups:

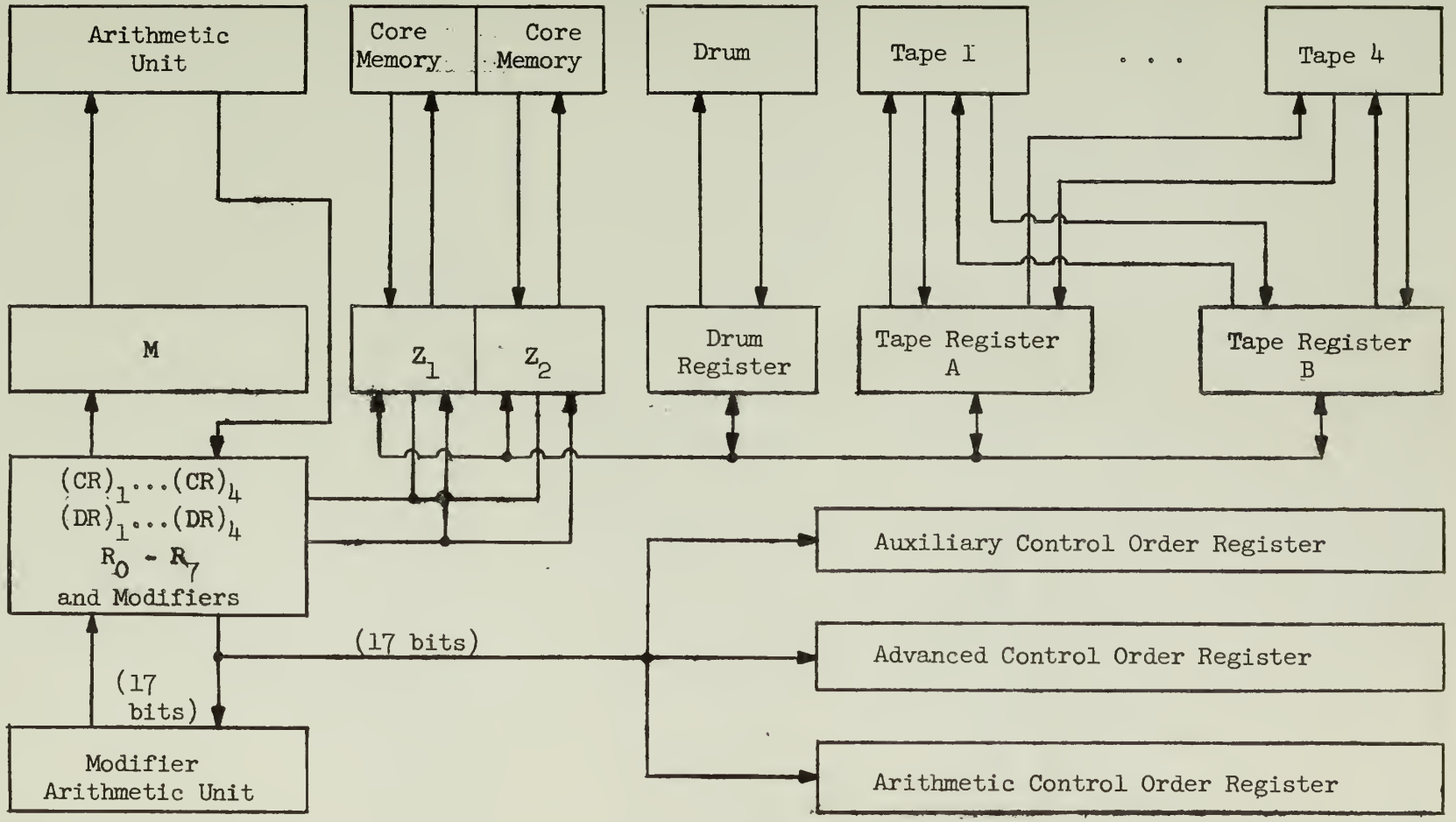
Type I (internal)

a) The accumulator overflows at the time of a store order.
b) The accumulator is overflowed.
c) Exponent overflow.
d) During standardization, more than a given number of shifts are executed. (This includes exponent underflow).
e) A change in a given position of the clock.

Type E (external)

a) Illegal order found by advanced control.
b) Drum finished last block transfer.
c) Tape unit finished last order.
d) Drum error.
e) Tape error.
f) A change in a given position of the clock.
g) A certain manual switch on the console has been depressed.

A register of equal length called the mask can be set by the programmer. Whenever the mask has a "1" in a position where the interrupt register is a "1", the execution of the program is temporarily halted, and a new sequence of


Arithmetic Unit
Core Memory
Core Memory
Drum
Tape 1
. . .
Tape 4
M
$Z_1$
$Z_2$
Drum Register
Tape Register A
Tape Register B
$(CR)_1 \ldots (CR)_4$
$(DR)_1 \ldots (DR)_4$
$R_0$ - $R_7$
and Modifiers
Auxiliary Control Order Register
(17 bits)
Advanced Control Order Register
(17 bits)
Modifier Arithmetic Unit
Arithmetic Control Order Register

instructions obeyed which will determine the cause of the interruption, and do what is necessary to rectify it. Next, the control will be returned to the place it left off, except, of course, in those cases where the interrupt was due to an incorrect program.

### 1.8.2 Clock Register

A clock register will be provided which can be set by the machine operator (normally once a day) but not by the programmer, and will be used for automatic log-keeping by the input routine for timing the computer by engineering routines, and for various other purposes in automatic code checking, real time problems, etc. One stage (perhaps 4 to 10 c. p. s.) will be connected to one position of the interrupt register.

### 1.8.3 Drum Position Register

Communication with the drum will be by blocks of fixed length, probably 128 words. There will be a counter associated with the drum, which indicates which block of words is currently under the magnetic heads. Since this is only 6 to 8 bits, and would be useful for some programs to use the drum efficiently, this register will be accessible to the programmer.

### 1.8.4 Indicator Register

The interrupt register, clock register, and drum position register together form one 52-bit register, called the indicator register, which is addressable. Only part of it, the interrupt register, may be written into, however.

## 2. THE MAKE-UP OF WORDS AND INSTRUCTIONS

### 2.1 Words

A 52-bit word may represent one of the following: a 45-bit fixed point number, a packed floating point number consisting of a 45-bit fractional part and a 7-bit, base 4, exponent or three 17-bit control groups, together with one spare bit.

### 2.2 Instructions

An instruction consists of one or two control groups. A long instruction (two control groups) can consist of two consecutive control groups in one word, or the last control group of one word and the first control group of the next word. Some instructions always require one control group, some always require two, and some may be short or long depending on the first 17 bits.

### 2.3 Control Groups for Arithmetic Instructions

The first control group of an instruction consists of 8 function bits and 9 bits which define the operand in the case of a short instruction. For arithmetic instructions, the first digit of the 9 specifies whether the remaining 8 bits are split into 2 category bits and 6 address bits or 3 category bits and a 5-bit group defining a modifier register.

The 4 categories for an address are:

a) fixed address in the core memory,

b) a relative address in the core memory referred to the control counter,

c) an integer to be used as an operand,

d) a fast access transistor register.

When a modifier is specified 5 of the 8 categories are used as follows:

1. The specified modifier is taken as the core address to be used with the instruction.

2. The specified modifier is used as the core address to be used in the instruction and the modifier is increased by one after it has been used as the address.

3. This is similar to (2) except the modifier is decreased by 1.

4. This is the same as (2) except the modifier is increased by the contents of the next higher numbered modifier register.

5. A long modified instruction, the next control group is interpreted as a core address added to the contents of the specified modifier to obtain the actual address used in the instruction. This is the only long arithmetic instruction. Two other categories are used to denote modification by any two of the first 8 modifier registers. The last category is spare.

## 2.4 Jump Instructions (Control Transfer Instructions)

We distinguish two classes of jump instructions modifier-conditional or M-conditional jumps in which the decision of whether or not to jump depends on the state of a modifier, and the remaining jumps (arithmetic or A-conditional, and jumps dependent on the state of some input-output unit or external device). The last 9 bits in the first control group are divided into 4 bits which specify an address, and 5 bits which specify the modifier, for M-conditional jumps or the jump condition (one of 32) for the remaining jumps. The address is in the range $\pm$ 7 relative to the control counter, and the value -8 denotes that a long instruction is present: the address is given by the next control group instead. The function digits of a jump instruction specify which control group inside a word is to be executed if the condition holds. Therefore, there are three versions of each jump instruction.

Since a very high proportion of jump instructions refer to memory locations near the location of the current instruction, it is expected that most jump instructions will be short.

## 2.5 Modifier Arithmetic Instructions

A modifier arithmetic instruction has only 7 function bits, one of which specifies whether the address is used as an operand or as the location where the operand is to be found. The remaining 10 bits are split 5 and 5. The second 5 specify the modifier register to be changed, and the first 5 specify the address (an integer from 0 to 30 or a modifier in the case of a short instruction). A long instruction, denoted by 31 in the address position uses the next control group as either the operand or the core address of the word whose least significant control group is to be used as the operand.

## 3. THE OPERATION OF ARITHMETIC CONTROL AND ADVANCED CONTROL

### 3.1 Introduction

The problem of designing interlocks between these controls so that the machine will operate efficiently is not simple. We begin by listing a number of design objectives and problems that may arise.

a) Advanced control has the primary responsibility for keeping the core memory busy as much as possible. For this purpose it has 8 buffer registers: 4 for instructions [$(CR)_1$, $(CR)_2$, $(CR)_3$, $(CR)_4$] and 4 for operands and results [$(DR)_1$, $(DR)_2$, $(DR)_3$, $(DR)_4$]. Pre-fetched words of instructions (called control words) are placed in the control registers (CR) in the same position, modulo 4, that they occupy in the core memory. Data words (operands and results) are placed in data registers (DR) in cyclic order. Since a new word placed in any one of these buffers will overwrite the previous contents, it is necessary to establish that arithmetic control no longer needs the previous contents.

b) In well-written programs for this computer, a very high fraction of the instructions executed are short, modified instructions with automatic counting in the modifier referred to. That is, whole addresses are commonly held in modifiers and usually incremented each time they are used. Since advanced control uses the address to obtain a word from the core memory, and arithmetic control does not need the address but merely the word, advanced control performs all exclusively-modifier operations.

c) Core memory write operations are fundamentally different from read instructions in several ways. When advanced control encounters a write instruction, the number to be written has not yet been computed by arithmetic control. It would be too time-consuming for advanced control to wait for the computation, so the address of the write instruction is saved, and advanced control proceeds. The address must be saved since, in general, it cannot be computed by arithmetic control because it may depend on a modifier whose contents will have changed. Advanced control is prohibited from reading from the same core address as an unexecuted write instruction.

d) Advanced control uses the memory at a fairly steady rate, whereas arithmetic control spends most of its time multiplying and dividing, and executes the intervening instructions very quickly. This means that most core references are overlapped with multiplications and divisions, most counting and address modification is overlapped with core memory time, and cases of conflict between advanced control and arithmetic control over the use of the buffer registers is less common than would be expected if references occurred at random times.

e) Occasionally, modifiers are set by arithmetic control, for example:

a table look-up requires part of the accumulator to be copied to a modifier;

one instruction copies the accumulator exponent into a modifier register;

modifiers may be used for packing and unpacking words, and also as additional fast one-word registers.

When advanced control encounters such a modifier use, it sets an indicator (reset by arithmetic control) to prevent its use by advanced control until it has been used by arithmetic control.

f) Unconditional and modifier-conditional jumps are executed by advanced control. Arithmetic conditional (A-conditional) jumps are also executed by advanced control, but only when arithmetic control has caught up to advanced control. Advanced control can get one conditional jump ahead of arithmetic control, and it records whether or not the jump was executed. Normally, advanced control is ahead of arithmetic control, and the main interlock is to prevent arithmetic control from passing advanced control. However, during the execution of short loops of instructions held in fast access registers, two other situations can occur if advanced control gets far ahead of arithmetic control. First, advanced control can scan one entire loop ahead of arithmetic control, and get one lap ahead. Secondly, advanced control can actually process a short loop twice before arithmetic control enters the loop. These problems are solved by permitting arithmetic control to apparently pass advanced control if advanced control has executed a conditional jump instruction which advanced control has not executed.

g) When a program interrupt is required, advanced control is halted at the end of the current instruction, and arithmetic control is allowed to catch up and execute that instruction also before the interruption is performed. In case the current instruction is of the type "modify the next instruction by the contents of a modifier in addition to any other modification called for", advanced control is not halted until the end of the first instruction following such a "modify" instruction. The interruption causes the control counter (word and position) to be stored into a fixed place in the core memory, control to be transferred to another fixed place, and a further indicator to be set in the indicator register which prevents any further interruptions until the register has been re-set by the program.

h) One question not yet decided is whether fully automatic interlocks should be provided to prevent the program from altering or using storage locations involved in a block transfer between drum or tape units and the core memory. The present plan is that the programmer can test the auxiliary storage unit to see whether it is still busy, or he can arrange for a program interruption to take place on completion of the current block transfer. Two possible further interlocks are being considered. The first would divide the core memory into fixed blocks, and prevent reference to any block referred to in a current block transfer, while the second would retain two addresses for each block transfer--the address of the latest word used and of the last word in the block, and would check that no address used by advanced control lies between such a pair of addresses.

## 3.2 Details of the Control

The controls of the computer will very likely be designed to be speed-independent. However, to give some idea of the feasibility and complexity of the controls, and to make the rules referred to in a)-h) more precise, a possible design will be described.

The following additional equipment is required to sequence the two controls:

$D_1$ is a 4-bit control counter which holds the address inside $(CR)_1$.., $(CR)_4$ of the control group currently being executed by arithmetic control. The first 2 bits of $D_1$ specify which (CR) the group came from, and the second 2, which may take on only the values 00, 01, or 10, specify the control group inside of the word.

$D_2$ is a 4-bit control counter, similar to $D_1$, used by advanced control.

$D_3$ is a 15-bit control counter which gives the word and position in the core memory of the control group currently being executed by advanced control. $D_2$ is actually the rightmost 4 bit positions of $D_3$. This establishes the correspondence that a control word is buffered in the same location modulo 4 in the (CR) memory that it occupies in the core memory.

$A_z$ is a 14-bit register. One bit $a_z^*$ is set to 1 when advanced control encounters a core memory write instruction, and the address from that instruction is copied into the remaining 13 bits of $A_z$. The extra bit is set to 0 when the number has been computed and stored in the core memory. If advanced control encounters a write instruction while $a_z^* = 1$, it must wait until the previously called-for write has been performed ($a_z^* = 0$) before setting $A_z$ again and proceeding. Similarly, if $a_z^* = 1$, all addresses are compared with the address held in $A_z$. In case an equal address is found, an interlock prevents the word being obtained from the memory before it is even stored there.

$F_1$ is a flipflop set to 1 when advanced control executes a conditional jump instruction (whether it jumps or not) and set to 0 when arithmetic control executes a conditional jump instruction.

$F_2$ is a flipflop used by advanced control to mark whether a conditional jump instruction (indicated by $F_1 = 1$) caused a transfer of control or not (1 or 0).

$G_1$ is a 2-bit counter which holds the address in $(DR)_1$,.., $(DR)_4$ of the next data register to be used by arithmetic control.

$G_2$ is a 2-bit counter which holds the address in $(DR)_1$..,$(DR)_4$ of the next data register to be used by advanced control. Initially $G_1$ and $G_2$ are set equal.

$H_1$, $H_2$, $H_3$, $H_4$ are 2-bit indicators for $(DR)_1$.., $(DR)_4$. The first bit of an indicator is set to 1 by advanced control when an operand is placed in the register, and cleared to 0 by arithmetic control when the operand is sent to the arithmetic unit. When advanced control encounters a write instruction, it sets the second bit of the indicator corresponding to the next available (DR) to a 1. When arithmetic control places the relevant work in the (DR), it sets the first bit to 1. The combination 11 in the indicator causes the word to be written in the core memory at the next available cycle, and the indicator is then cleared to 00. Advanced control cannot read a word from the memory into a (DR) until its indicator has become 0, indicating that arithmetic control has used the previous word, or that the memory has stored the previous word.

$K_0$, $K_1$..., $K_7$ are 1-bit indicators, one for each of registers $R_0$,..., $R_7$. When advanced control encounters a use by arithmetic control of one of these registers (either a word or a modifier contained inside of that word) it sets the corresponding indicator to 1. Thereafter advanced control is prevented from using or changing any part of that register until arithmetic control has used the register, and has re-set the indicator to 0.

$M_1$, $M_2$, $M_3$, $M_4$ are 2-bit indicators for $(CR)_1$.., $(CR)_4$, used to define the significance of the words in the control registers. Their purpose is to minimize instruction reads from the core memory. The first bit of an indicator is 1 if the word in the corresponding (CR) is one of the three next higher addressed words relative to the control counter $D_3$. The second indicator bit is a 1 if the word is the current word or one of the three next lower addressed words relative to the control counter.

## 3.3 Sequencing of Instructions

When the computer is started, $D_3$ is set to some initial value, $D_1$ is set equal to $D_2$, and $a_2^*$, $F_1$, $G_1$, $G_2$, $H_1$..$H_4$, $K_0$..$K_7$, $M_1$..$M_4$ are all cleared to 0. Since $F_1$ is 0 and $D_1$ and $D_2$ agree, arithmetic control is caught up with advanced control and must wait. $D_2$ is not changed, even for a long instruction, until advanced control has obtained any operand required by arithmetic control. If $F_1$ is 1 but $D_1$ and $D_2$ agree, arithmetic control can execute the instruction since advanced control is preparing for the next execution of that instruction by arithmetic control.

For definiteness, suppose that the value of $D_3$ is such that the current word should be placed in $(CR)_1$. Since, at the start, the second bit of $M_1$ is 0, the word is not in $(CR)_1$ already, and must be read from the memory. Advanced control reads the word from the core address given by $D_3$ into $(CR)_1$ and then sets the second bit of $M_1$ to 1.

To obtain the next control group in sequence, advanced control counts once or twice on $D_3$ (according to whether current instruction was short or long). If the least significant 2 bits of $D_3$ are 01 or 10, the next control group in sequence is in the same word as the last used control group, so advanced control knows that $D_2$ defines the location in (CR) of the control group. If the least significant 2 bits of $D_3$ are 0's, a new word is referred to, which may or may not be in the (CR) specified by $D_2$. Suppose $(CR)_2$ is the new register involved. Advanced control copies the first bit of $M_2$ into the second bit position of $M_2$, and then clears the first bit to 0. If now the second bit of $M_2$ is 1, the word is in $(CR)_2$ already, so no core reference is required. If not a 1, advanced control waits (if necessary) until the first 2 bits of $D_1$ and $D_2$ disagree, indicating that arithmetic control is not executing old instructions from $(CR)_2$. Advanced control then reads the word from the core memory into $(CR)_2$ and proceeds to execute it.

Arithmetic control obtains the next control group in sequence by increasing $D_1$ by 1 or 2 according to whether the current instruction is short or long. If $F_1 = 1$, arithmetic control executes the next control group immediately. If $F_1 = 0$, arithmetic control waits (if necessary) until $D_1$ and $D_2$ are no longer identical--indicating that advanced control has now processed the current control instruction and then arithmetic control executes the current instruction.

Advanced control executes jump instructions. For an arithmetic conditional jump or a jump to a modified address, it waits until $D_1 = D_2$ before making the test. For any conditional jump it must wait until $F_1 = 0$ which means that arithmetic control has performed a previous conditional jump. If the jump is not executed because the condition did not apply, advanced control sets $F_1 = 1$, $F_2 = 0$ and proceeds to the next control group in sequence, as described before. If the jump is executed the new address is compared with the old contents of $D_3$

(13-bit subtraction). If the result is in the range -6 to 6 special considerations apply since $M_1..M_4$ must be updated. Otherwise, advanced control waits until arithmetic control has executed all control groups in the (CR) referred to, then reads the word from the memory, sets $D_3$ to the new address, clears $M_1..M_4$ to 0 and sets the second bit of the M defined by $D_2$ to 1. $F_1$ and $F_2$ are set to 1. Arithmetic control is considered to have executed an A-conditional jump when it has supplied the condition to advanced control. If the jump is to a modified address, advanced control sets $D_1$ equal to $D_2$.

In case the jump address differed from the address already in $D_3$ by one of 1, 2, ..., 6 advanced control proceeds forward cyclically through 1,2,...,6 of the M indicators, performing on each one in turn the operation "first bit to second bit; clear first bit". It then omits the core read if the second bit of the last changed M is a 1. In case the jump address differed from the address already in $D_3$ by one of -1, -2,...,-6, advanced control proceeds backward cyclically through 1,2,...,6 of the M indicators, performing on each one in turn the operation "second bit to first bit; clear second bit". It then omits the core read if the first bit of the last changed M is 1.

A program interrupt condition causes advanced control to halt before obeying the next instruction. When arithmetic control has caught up, $D_3$ is stored automatically and an unconditional jump is performed to a fixed interrupt location.

## 3.4 Execution of Instructions

The core memory, the fast transistor registers, the arithmetic unit and the modifier arithmetic unit are all time-shared between arithmetic control and advanced control. An interlock prevents each one from being used by one control while the other is using it.

For most instructions, modifier arithmetic is performed by advanced control. However some instructions have the effect of setting one or more modifiers from the memory or from an arithmetic result. In this case, advanced control sets one of the indicators $K_0..$, $K_7$, and is permitted to proceed, provided it does not again execute an instruction involving this register until arithmetic control has used the register and re-set the indicator.

Operands to be used by the arithmetic unit, or results to be sent from the arithmetic unit to the core memory, are assigned (DR) registers in the order of occurrence, cyclically. The interlock to prevent advanced control from setting a (DR) earlier than it should has been described under the section defining $H_1$.., $H_4$.

4. ORDER CODE

## 4.1 Floating Point Arithmetic

The accumulator (register A,Q) define one double precision number, which will be referred to in this discussion as A,Q. The first step of multiplication is to transfer A (normalized and rounded) via R to Q. During multiplication, the multiplier is held in Q, the multiplicand in M, and the product (double precision, unrounded) is generated in the accumulator by a sequence of additions, subtractions, and right shifts. Since addition and subtraction are also double precision operations, working to full double precision is made very easy. For example, to add two double precision numbers, each represented by two words, it is merely necessary to add the 4 operands in any sequence, and then store the result.

The "clear add" and "clear subtract" instructions clear the double length accumulator to zero, before adding or subtracting. The representation of a number in the accumulator is to double precision, but not necessarily normalized. Whenever an addition or subtraction would involve a loss of precision in the addend or subtrahend because its exponent is too small, an attempt is made first to normalize the accumulator. In summary, the maximum precision obtainable with a double length accumulator is obtained. The precision is identical to that which would have been obtained if the accumulator had been normalized at every stage of the calculation.

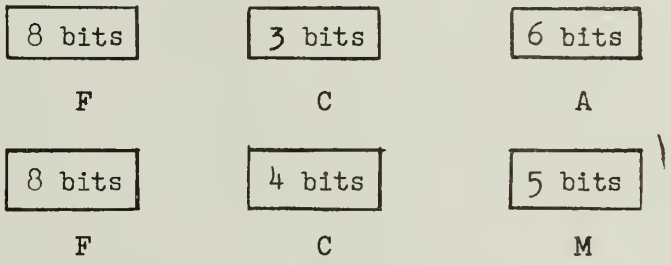
When a number is to be stored from the accumulation, A is assimilated and AQ is standardized. The number to be stored is rounded to single precision. Since a very common occurrence in floating point addition is to add two numbers whose exponents are nearly equal, it is frequently the case that precisely 1/2 in the least significant retained digit must be rounded up or down. In this case the round-off is such that the least significant digit of the result is zero.

Double precision addition of products (for example, calculating the scalar product of 2 vectors) is facilitated by the instructions "prepare to add product" and "add product". The prepare-to-add product instruction first stores both halves of the accumulator in fast access registers $R_6$ and $R_7$, and

then clears the accumulator and adds the number from memory to the accumulator. The add-product instruction first multiplies the number from memory by the accumulator and then adds both $R_6$ and $R_7$ to the accumulator. The effect of these two instructions is to add the product of their operands, double precision, to the number originally in the accumulator.

The store-and-subtract instruction causes the accumulator to be normalized, rounded off, stored and the result subtracted. This has the effect of storing the most significant half of the accumulator, and leaving the least significant half only in the accumulator.

An arithmetic instruction of 17 bits is made up in one of the two following ways:




F stands for function, C for category, A for address, and M for modifier. The first category bit distinguishes whether there are 3 or 4 category bits (0 or 1 respectively). One of the categories beginning with 1 designates a long instruction. In this case the next control group is a 17-bit address to be added to the designated modifier to form a core memory address.

## 4.2 Floating Arithmetic Order Code

1. Clear add*
2. Hold add
3. Clear subtract
4. Hold subtract
5. Clear add absolute value

---

* This does not normalize the accumulator even if the number from memory did not have a normalized fractional part.

6. Hold add absolute value

7. Clear subtract absolute value

8. Hold subtract absolute value

9. Positive multiply

10. Negative multiply

11. Add product

12. Subtract product

13. Positive divide

14. Negative divide

15. Prepare to add product

16. Integer division: remainder in AQ, quotient in $R_7$

17. Take absolute value of AQ and subtract the absolute value of the number from the memory

18. Hold add and store the result back into the same storage register

19. Hold subtract and store the result

20. Exchange with memory location

21. Store

22. Replace AQ by the negative rounded value and store

23. Store and subtract result from the accumulator

24. Clear add to the exponent

25. Hold add to the exponent

26. Clear subtract from the exponent

27. Hold subtract from the exponent

The following 3 orders involve both the accumulator and the modifier arithmetic unit, and have the same make-up of digits as the M-arithmetic instructions:

28. Transfer the exponent to a modifier register

29. Multiply by integer, round accumulator to single precision, and transfer the unrounded integer part of the result to specified modifier, leave fraction in accumulator.

30. Add n to the exponent, round AQ to single precision, and transfer the unrounded integer part of the result to specified modifier, leave fraction in accumulator.

## 4.3 Fixed Point Arithmetic

A fixed point number held in the memory has zero exponent and is not necessarily normalized. The first fixed point instruction (which converts from floating point to this representation) is

31. Store fixed point (Shift the accumulator, adjusting its exponent, until its exponent becomes equal to zero. Round-off and store the result.

It is possible, with the aid of this instruction, to do fixed point arithmetic in floating point, converting to fixed point only when going to a memory register. Floating point division instructions arrange to normalize the divisor first, before divide. However, several other orders are provided to make fixed point even easier.

32. Store integer part (but with zero exponent).

33. Store and subtract. This leaves the least significant half of the accumulator in the most significant end, with a zero exponent also.

34. Store the positive fractional part, and replace AQ by the integer part but with zero exponent.

35. Add to Q performing all carries.

36. Subtract from Q.

37. Store Q without rounding off A.

38. Store A without rounding off.

The following instructions while not arithmetic in nature, have the same structure (make-up of bits) as arithmetic instructions:

39. Add the address of this instruction to the address of the next instruction.

40. Add to the address of the next instruction the 17 least significant bits of this operand.

41. Set the M-digits of the next instruction equal to the 5 least significant bits of the address of this instruction.

42. Set fast register equal to the contents of the core location given by the next address (this is a long instruction).

43. Store contents of fast register in core location specified by the next address (this is a long instruction).

44. Set modifier to 0, +1 or -1 according to whether the accumulator is zero, positive, or negative.

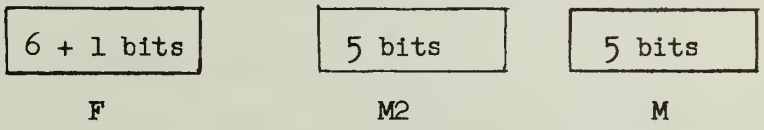
45. Set modifier to -1, 0, +1 according to whether the accumulator exponent is overflowed, in range, or underflowed.

46. Set modifier to 0, -1 according to whether the accumulator is known to be exact or may be inexact (i.e., it would require more than 90 bits to specify the result of an addition or subtraction which has been performed since the accumulator was last cleared, or, if exponent underflow has occurred.)

47. Round and store the accumulator, and place in modifier No. 1 the number of shift required to normalize the accumulator.

## 4.4 Modifier Arithmetic

The make-up of an M-arithmetic instruction is




where M is the designated modifier and M2 is either a 5-bit non-negative operand or the address of a second modifier whose contents is to be used as

an operand. However, if $M_2$ = 11111, this is a long instruction, and the next control group represents either a 17-bit operand or the core address of a word whose least significant 17 bits comprise the operand.

48. Clear add.

49. Hold add.

50. Clear subtract.

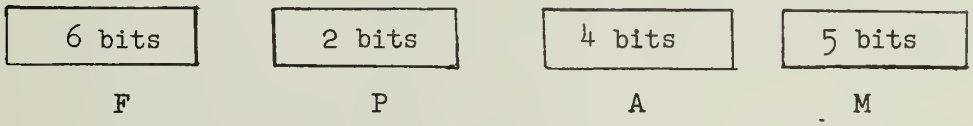
51. Hold subtract.

52. Cyclic binary left shift.

See also orders 60 and 61.

## 4.5 Jump Instructions

The make-up of a jump instruction is:




where P is the position (00, 01, or 10) inside the word referred to, A is a 4-bit relative address, and M represents either a modifier or one of 32 input-output or arithmetic conditions. One combination, A = 1000 designates a long instruction, and the address consisting of the next control group represents the fixed core location referred to.

53. Unconditional

54. Unconditional, set modifier to value of control counter.

55. Jump if modifier contents = 0.

56. Jump if modifier contents $\neq$ 0.

57. Jump if modifier contents $\geq$ 0.

58. Jump if modifier contents $<$ 0.

59. Arithmetic conditional, conditions given by M digits are:

(1) Accumulator $\geq 0$

(2) Accumulator $< 0$

(3) Accumulator $= 0$

(4) Accumulator $\neq 0$

(5) Exponent $\geq 0$

(6) Exponent $< 0$

(7) Exponent $= 0$

(8) Exponent $\neq 0$

(9) Accumulator result imprecise (c.f. order 46)

(10) Precise

(11) Floating point overflow (Re-set overflow)

(12) Floating point non-overflow (re-set overflow)

(13) Exponent underflow

(14) Exponent not underflowed

60. Decrease contents of modifier by 1 and jump unless it is now zero.

The following orders are strictly modifier arithmetic but have a digit make-up like jump instructions.

61. Store modifier in word A position P

62. Set modifier equal to contents of word A, position P.

## 4.6 Logical Orders

A logical word comprises the fractional part of a fixed point number, whose exponent is zero. Therefore, the exponent would not be affected if AND, OR, or EXCLUSIVE OR were applied to the full word instead of merely 45 bits. It appears that the OR operation will be possible at very little extra cost on read-out from the flow gating memory. It would also be desirable to have the two other operations somewhere, especially EXCLUSIVE OR. However the details of these orders have not been settled.

## 4.7 Input-Output and Auxiliary Storage

The details of the printer, paper tape, magnetic tape, and drum orders have not been settled.